Introduction to Neural Prediction

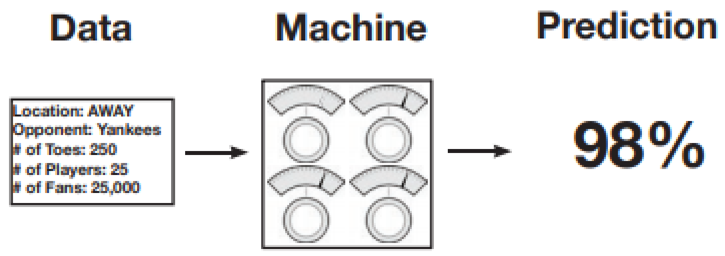
Forward Propagation

I try not to get involved in the business of prediction. It's a quick way to look like an idiot. — WARREN ELLIS

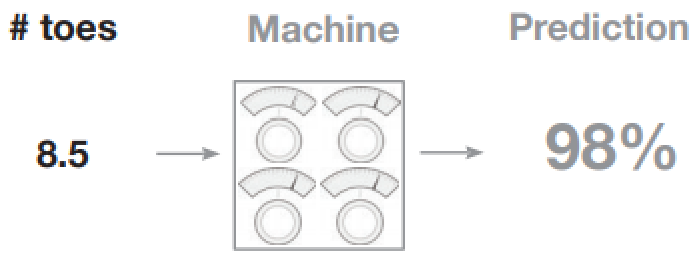
Chapter 3 I 22 Introduction to Neural Prediction Location:

Step 1: Predict This chapter is about "Prediction"

In the previous chapter, we learned about the paradigm: "Predict, Compare, Learn". In this chapter, we will dive deep into the fi rst step, "Predict". You may remember that the predict step looks a lot like this.

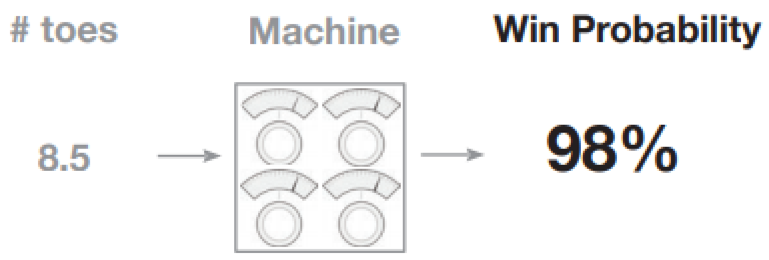


In this chapter, we're going to learn more about what these 3 diff erent parts of a neural network prediction really look like under the hood. Let's start with the fi rst one, the Data. In our fi rst neural network, we're going to predict one datapoint at a time, like so.

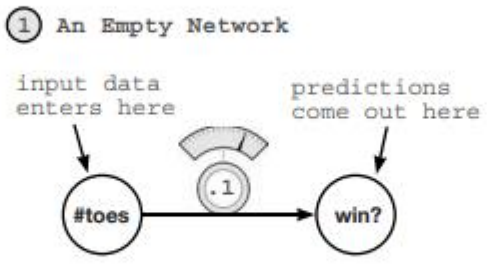


# toes Machine Prediction 8.5 98% Later on, we will fi nd that the "number of datapoints at a time" that we want to process will have a signifi cant impact on what our network looks like. You might be wondering, "how do I choose how many datapoints to propagate at a time?" Th e answer to this question is based on whether or not you think the neural network can be accurate with the data you give it. For example, if I'm trying to predict whether or not there's a cat in a photo, I defi nitely need to show my network all the pixels of an image at once. Why? Well, if I only sent you one pixel of an image, could you classify whether the image contained a cat? Me neither! (Th at's a general rule of thumb by the way. Always present enough information to the network, where "enough information" is defi ned loosely as how much a human might need to make the same prediction).

Let's skip over the network for now. As it turns out, we can only create our network once we understand the shape of our input and output datasets (For now, shape means "number of columns" or "number of datapoints we're processing at once"). For now, we're going to stick with the "single-prediction" of "likelihood that the baseball team will win".



Ok, so now that we know that we want to take one input datapoint and output one prediction, we can create our neural network. Since we only have one input datapoint and one output datapoint, we're going to build a network with a single knob mapping from the input point to the output. Abstractly these "knob"s are actually called "weight"s, and we will refer to them as such from here on out. So, without further ado, here's our fi rst neural network with a single weight mapping from our input "#toes" to output "win?"

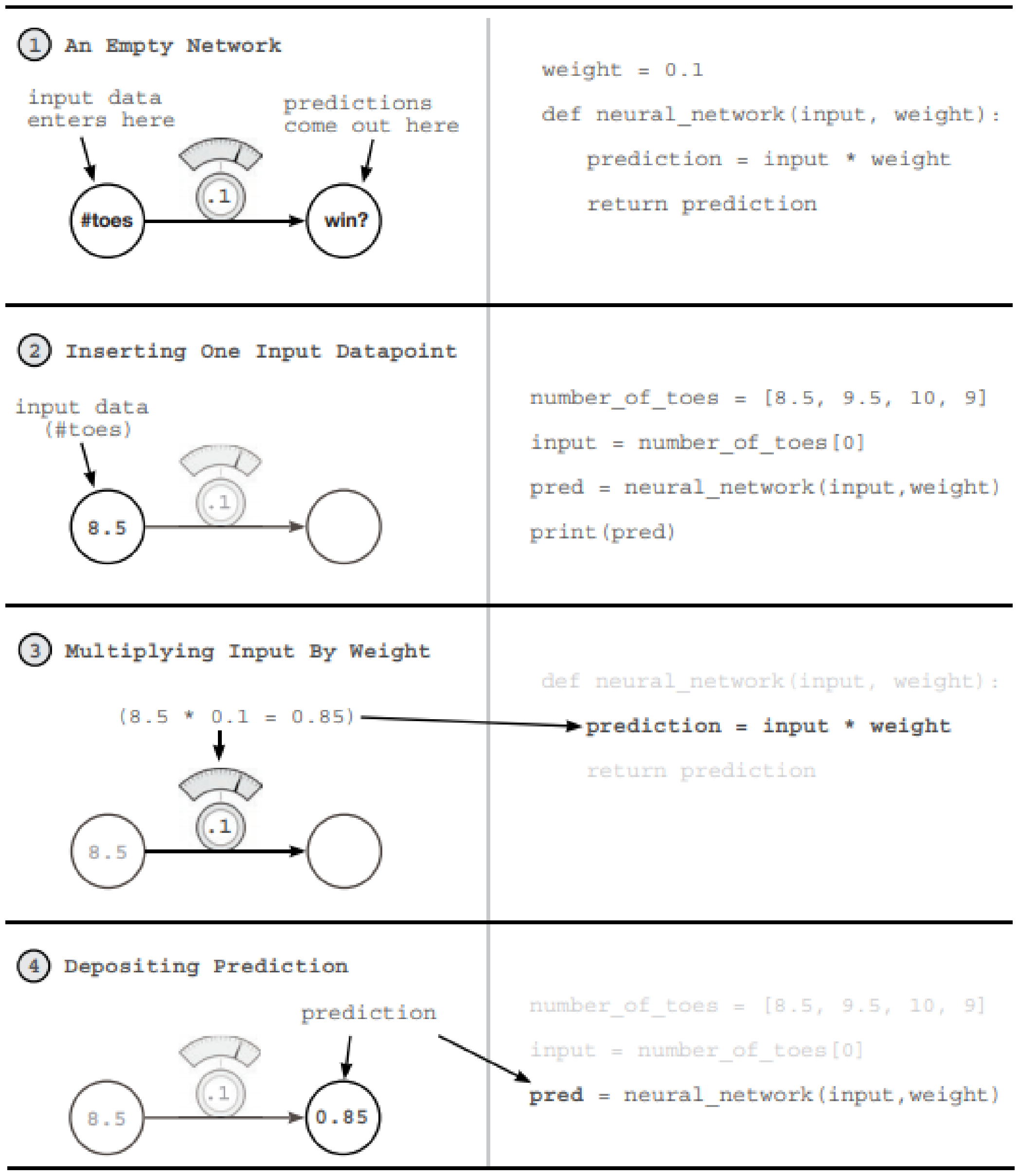


As you can see, with one weight, this network takes in one datapoint at a time (average number of toes on the baseball team) and outputs a single prediction (whether or not it thinks the team will win).

Prediction A Simple Neural Network Making a Prediction

Let's start with the simplest neural network possible.

Let's start with the simplest neural network possible.



```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)

print(pred)
```

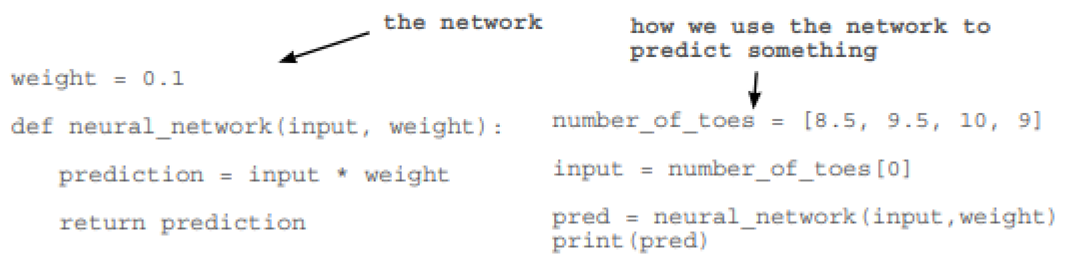
```
def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```

What is a Neural Network?

What is a Neural Network?

Open up a Jupyter Notebook and run the following:



```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
print(pred)
```

You just made your first neural network and used it to predict! Congratulations! The last line prints the prediction (pred). It should be 0.85. So what is a neural network? For now, it's one or more weights which we can multiply by our input data to make a prediction.

What is input data?

It's a number that we recorded in the real world somewhere. It's usually some thing that is easily knowable, like today's temperature, a baseball player's batting average, or yesterday's stock price.

What is a prediction?

A prediction is what the neural network tells us given our input data such as "given the temperature, it is 0% likely that people will wear sweatsuits today" or "given a baseball player's batting average, he is 30% likely to hit a home run" or "given yesterday's stock price, today's stock price will be 101.52".

Is this prediction always right?

No. Sometimes our neural network will make mistakes, but it can learn from them. For example, if it predicts too high, it will adjust it's weight to predict low er next time and vice versa.

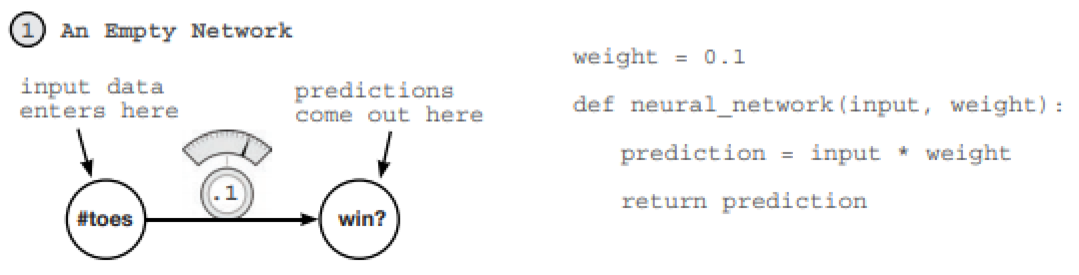
How does the network learn?

Trial and error! First, it tries to make a prediction. Then, it sees whether it was too high or too low. Finally, it changes the weight (up or down) to predict more accurately the next time it sees the same input.

What does this Neural Network do?

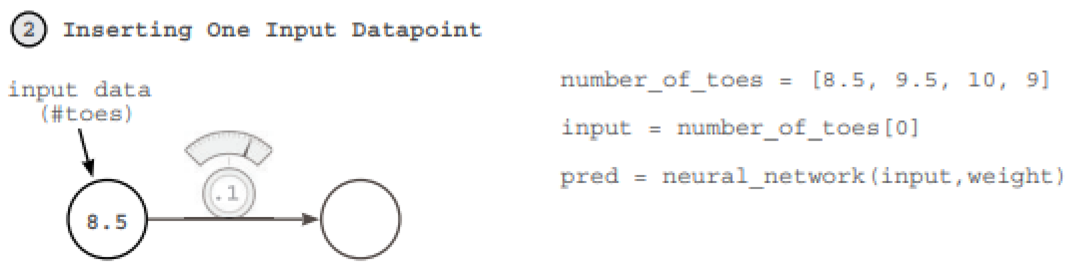
It multiplies the input by a weight. It "scales" the input by a certain amount.

On the previous page, we made our fi rst prediction with a neural network. A neural network, in it's simplest form, uses the power of multiplication. It takes our input datapoint (in this case, 8.5) and multiplies it by our weight. If the weight is 2, then it would double our input. If the weight is 0.01, then it would divide the input by 100. As you can see, some weight values make the input bigger and other values make it smaller.



```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

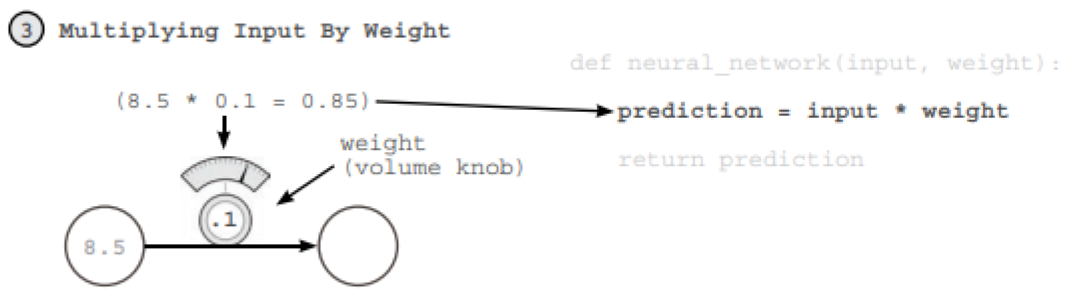
The interface for our neural network is really quite simple. It accepts an input variable as information, and a weight variable as knowledge and outputs a prediction. Every neural network you will ever see works this way. It uses the knowledge in the weights to interpret the information in the input data. Later neural networks will accept larger, more complicated input and weight values, but this same underlying premise will always ring true.



```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```

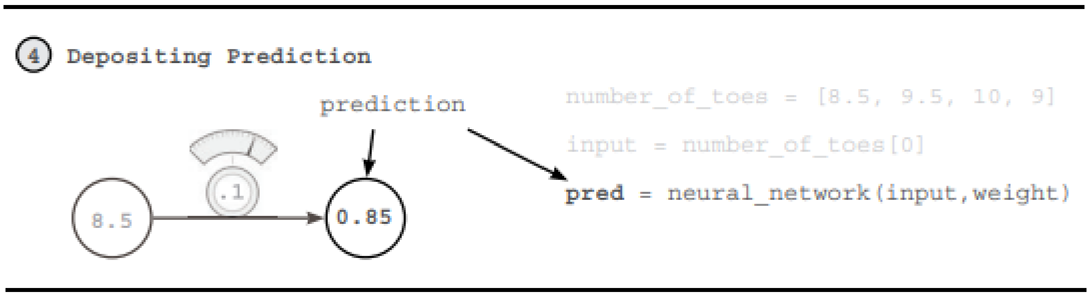
In this case, the "information" is the average number of toes on a baseball team before a game. Notice several things. Th e neural network does NOT have access to any information except one instance. If, aft er this prediction, we were to feed in number_of_toes[1], it would not remember the prediction it made in the last timestep. A neural network only knows what you feed it as input. It forgets everything else. Later, we will learn how to give neural networks "short term memories" by feeding in multiple inputs at once.

What does this Neural Network do?

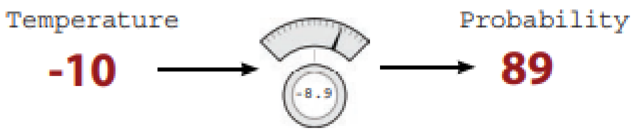


```
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

27 Another way to think about a neural network's weight is as a measure of sensitivity be tween the input of the network and its prediction. If the weight is very high, then even the tiniest input can create a really large prediction! If the weight is very small, then even large inputs will make small predictions. Th is sensitivity is very akin to volume. "Turning up the weight" amplifi es our prediction relative to our input. weight is a volume knob!


4 Depositing Prediction
prediction
8.5
.1
0.85
number_of_toes = [8.5, 9.5, 10, 9]
input = number_of_toes[0]
pred = neural_network(input,weight)

So in this case, what our neural network is really doing is applying a volume knob to our number_of_toes variable. In theory, this volume knob is able to tell us the likelihood that the team will win based on the average number of toes per player on our team. And this may or may not work. Truthfully, if the team had 0 toes, they would probably play terribly. However, baseball is much more complex than this. On the next page, we will present multiple pieces of information at the same time, so that the neural network can make more informed decisions. Before we go, neural networks don't just predict positive numbers either, they can also predict negative numbers, and even take negative numbers as input. Perhaps you want to predict the "probability that people will wear coats today", if the temperature was -10 degrees Celsius, then a negative weight would predict a high probability that people would wear coats today.

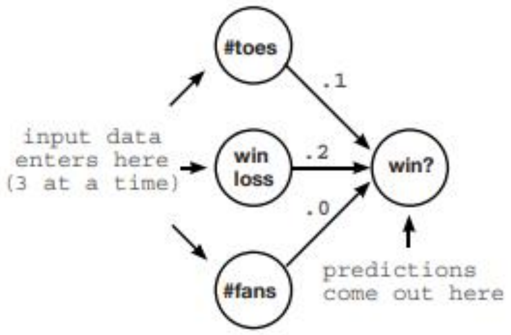


Making a Prediction with Multiple Inputs

Neural Networks can combine intelligence from multiple datapoints.

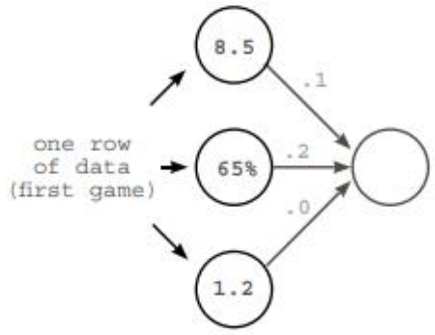
Our last neural network was able to take one datapoint as input and make one prediction based on that datapoint. Perhaps you've been wondering, "is average # of toes really a very good predictor?... all by itself?" If so, you're onto something. What if we were able to give our network more information (at one time) than just the "average number of toes". It should, in theory, be able to make more accurate predictions, yes? Well, as it turns out, our network can accept multiple input datapoints at a time. See the prediction below!

① An Empty Network With Multiple Inputs



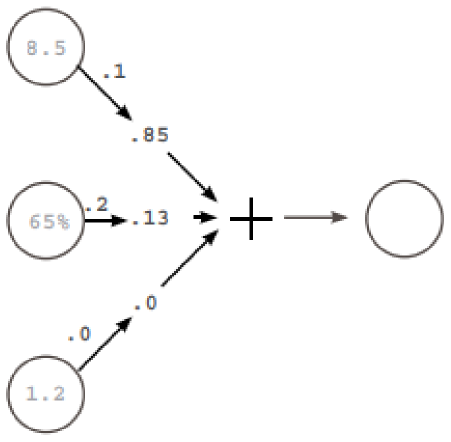
```
weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

② Inserting One Input Datapoint



```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */


toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

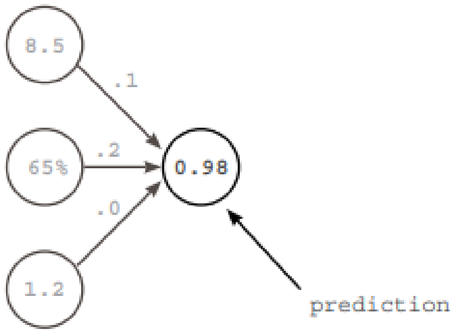
3 Perform a Weighted Sum of Inputs



```
def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred

def w_sum(a,b):

    assert(len(a) == len(b))

    output = 0

    for i in range(a):
        output += (a[i] * b[i])

    return output
```

```
                          local
 inputs   weights      predictions
( 8.50  *  0.1 )  =   0.85  = toes prediction
( 0.65  *  0.2 )  =   0.13  = wlrec prediction
( 1.20  *  0.0 )  =   0.00  = fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

      0.85      +       0.13       +       0.00      =      0.98
```

4 Deposit Prediction

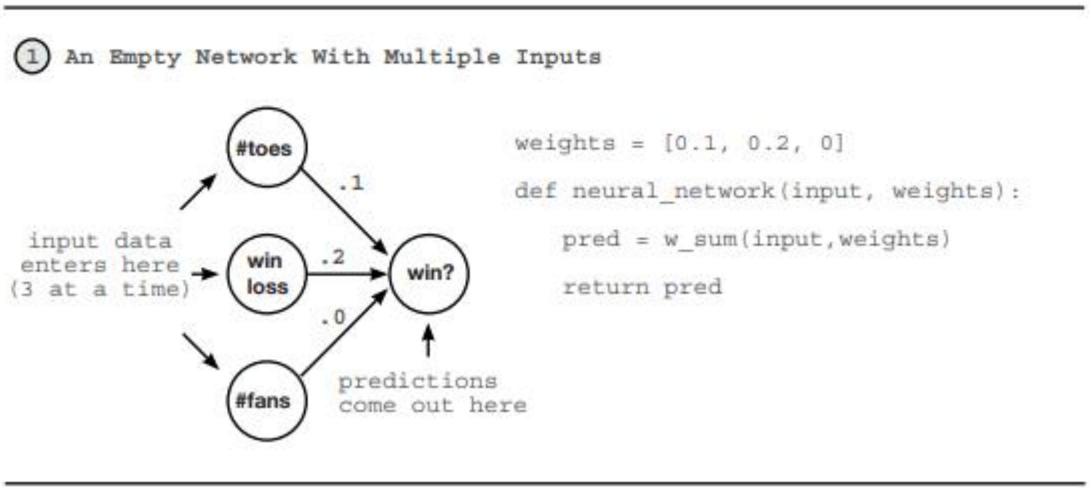


```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

Multiple Inputs - What does this Neural Network do?

It multiplies 3 inputs by 3 knob_weights and sums them. This is a "weighted sum".

At the end of the previous section, we came to realize the limiting factor of our sim ple neural network, it is only a volume knob on one datapoint. In our example, that datapoint was the average number of toes on a baseball team. We realized that in order to make accurate predictions, we need to build neural networks that can combine multiple inputs at the same time. Fortunately, neural networks are perfectly capable of doing so.



① An Empty Network With Multiple Inputs
input data enters here (3 at a time)
#toes
win loss
#fans
.1
.2
.0
win?
predictions come out here

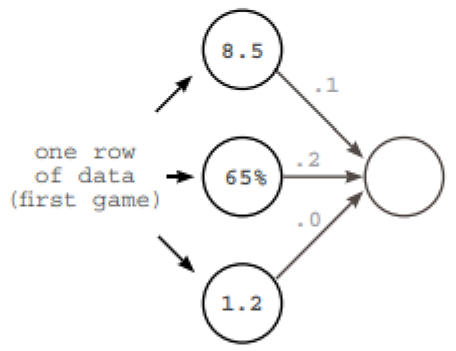
```
weights = [0.1, 0.2, 0]
def neural_network(input, weights):
    pred = w_sum(input,weights)
    return pred
```

In this new neural network, we can accept multiple inputs at a time per prediction. This allows our network to combine various forms of information to make more well informed decisions. However, the fundamental mechanism for using our weights has not changed. We still take each input and run it through its own volume knob. In other words, we take each input and multiply it by its own weight. The new property here is that, since we have mutliple inputs, we have to sum their respective predictions. Thus, we take each input, multiply it by its respective weight, and then sum all the local predictions together. This is called a "weighted sum of the input" or a "weighted sum" for short. Some also refer to this "weighted sum" as a "dot product" as we'll see.

A Relevant Reminder

The interface for our neural network is quite simple. It accepts an input variable as information, and a weight variable as knowledge and outputs a prediction.

(2) Inserting One Input Datapoint



```
/* This dataset is the current
status at the beginning of
each game for the first 4 games
in a season.

toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions) */


toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

# input corresponds to every entry
# for the first game of the season

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

This new need to process multiple inputs at a time justifi es the use of a new tool. Th is tool is called a vector and if you've been following along in your iPython notebook, you've already been using it. A vector is nothing other than a list of numbers. input is a vector and weights is a vector. Can you spot any more vectors in the code above (there are 3 more)? As it turns out, vectors are incredibly useful whenever you want to perform operations involving groups of numbers. In this case, we're performing a weighted sum between two vectors (dot product). We're taking two vectors of equal length (input and weights), multiplying each number based on its position (the fi rst position in input is multiplied by the fi rst position in weights, etc.), and then summing the resulting output. It turns out that whenever we perform a mathematical operation between two vectors of equal length where we "pair up" values according to their position in the vector (again... position 0 with 0, 1, with 1, and so on), we call this an elementwise operation. Th us "elementwise addi tion" sums two vectors. "elementwise multiplication" multiplies two vectors.

2 Inserting One Input Datapoint one row of data (first game) .1 .2 .0 8.5 65% 1.2 toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) /* This dataset is the current status at the beginning of each game for the first 4 games in a season. toes = current number of toes wlrec = current games won (percent) nfans = fan count (in millions) */ Being able to manipulate vectors is a cornerstone technique for Deep Learning. See if you can write functions that perform the following operations: def elementwise_multiplication(vec_a, vec_b) def vector_sum(vec_a) def elementwise_addition(vec_a, vec_b) def vector_average(vec_a) Th en, see if you can use two of these methods to perform a dot product! Challenge: Vector Math Licensed to Asif Qamar Chapter 3 I 32 Introduction to Neural Prediction 3 Perform a Weighted Sum of Inputs .1 .2 .0 8.5 65% 1.2 def neural_network(input, weights): pred = w_sum(input,weights) return pred def w_sum(a,b): assert(len(a) == len(b)) output = 0 for i in range(a): output += (a[i] * b[i]) return output (8.50 * 0.1) = 0.85 = toes prediction (0.65 * 0.2) = 0.13 = wlrec prediction (1.20 * 0.0) = 0.00 = fans prediction toes prediction + wlrec prediction + fans prediction = final prediction 0.85 + 0.13 + 0.00 = 0.98 inputs weights .85 .13 .0 local predictions Th e intuition behind how and why a dot product (weighted sum) works is easily one of the most important parts of truly understanding how neural networks make predictions. Loosely stated, a dot product gives us a notion of similarity between two

vectors. Consider the examples: a = [0, 1, 0, 1] b = [1, 0, 1, 0] c = [0, 1, 1, 0] d = [.5, 0,.5, 0] e = [0, 1,-1, 0] w_sum(a,b) = 0 w_sum(b,c) = 1 w_sum(b,d) = 1 w_sum(c,c) = 2 w_sum(d,d) = .5 w_sum(c,e) = 0 Th e highest weighted sum (w_sum(c,c)) is between vectors that are exactly identical. In contrast, since a and b have no overlapping weight, their dot product is zero. Perhaps the most interesting weighted sum is between c and e, since e has a negative weight. Th is negative weight cancelled out the positive similarity between them. However, a dot product between e and itself would yield the number 2, despite the negative weight (double negative turns positive). Let's become familiar with these properties. Licensed to Asif Qamar Multiple Inputs - What does this Neural Network do? 33 Some have equated the properties of the "dot product" to a "logical AND". Consider a and b. a = [0, 1, 0, 1] b = [1, 0, 1, 0] If you asked whether both a[0] AND b[0] had value, the answer would be no. If you asked whether both a[1] AND b[1] had value, the answer would again be no. Since this is AL WAYS true for all 4 values, the final score equals 0. Each value failed the logical AND. b = [1, 0, 1, 0] c = [0, 1, 1, 0] b and c, however, have one column that shares value. It passes the logical AND since b[2] AND c[2] have weight. This column (and only this column) causes the score to rise to 1. c = [0, 1, 1, 0] d = [.5, 0,.5, 0] Fortunately, neural networks are also able to model partial ANDing. In this case, c and d share the same column as b and c, but since d only has 0.5 weight there, the final score is only 0.5. We exploit this property when modeling probabilities in neural networks. d = [.5, 0,.5, 0] e = [-1, 1, 0, 0] In this analogy, negative weights tend to imply a logcal NOT operator, given that any positive weight paired with a negative weight will cause the score to go down. Furthermore, if both vectors have negative weights (such as w_sum(e,e)), then it will perform a double negative and add weight instead. Additionally, some will say that it's an OR after the AND, since if any of the rows show weight, the score is affected. Thus, for w_sum(a,b), if (a[0] AND b[0]) OR (a[1] AND b[1)...etc.. then have a positive score. Furthermore, if one is negative, then that column gets a NOT. Amusingly, this actually gives us a kind of crude language to "read our weights". Let's "read" a few examples, shall we? These assume you're performing w_sum(input,weights) and the "then" to these "if statements" is just an abstract "then give high score". weights = [1, 0, 1] => if input[0] OR input[2] weights = [0, 0, 1] => if input[2] weights = [1, 0, -1] => if input[0] OR NOT input[2] weights = [-1, 0, -1] => if NOT input[0] OR NOT input[2] weights = [0.5, 0, 1] => if BIG input[0] or input[2] Notice in the last row that a weight[0] = 0.5 means that the corresponding input [0] would have to be larger to compensate for the smaller weighting. And as I mentioned, this is a very very crude approximate language. However, I find it to be immensely useful when trying to picture in my head what's going on under the hood. This will help us significantly in the future, especially when putting networks together in increasingly complex ways. Licensed to Asif Qamar Chapter 3 I 34 Introduction to Neural Prediction 4 Deposit Prediction .1 .2 .0 8.5 65% 1.2 0.98 toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) prediction So, given these intuitions, what does this mean when our neural network makes a prediction? Very rougly speaking, it means that our network gives a high score of our inputs based on how similar they are to our weights. Notice below that "nfans" is completely ignored in the prediction because the weight associated with it is a 0. Th e most sensitive predictor, in fact, is "wlrec" because its weight is a 0.2. However, the dominant force in the high score is the number of toes ("ntoes") not because the weight is the highest, but because the input combined with the weight is by far the highest. A few more points that we will note here for further reference. We cannot shuffl e our weights. Th ey have specifi c positions they need to be in. Furthermore, both the value of the weight AND the value of the input determine the overall impact on the fi nal score. Finally, a negative weight would cause some inputs to reduce the fi nal prediction (and vise versa). Licensed to Asif Qamar Multiple Inputs - Complete Runnable Code 35 Multiple Inputs - Complete Runnable Code The code snippets from this example come together as follows. toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) print(pred) def w_sum(a,b): assert(len(a) == len(b)) output = 0 for i in range(a): output += (a[i] * b[i]) return output weights = [0.1, 0.2, 0] def neural_network(input, weights): pred = w_sum(input,weights) return pred We can create and execute our neural network using the following code. For the purposes of clarity, I have written everything out using only basic properties of Python (lists and numbers). However, there is a better way that we will start using in the future. There is a python library called "numpy" which stands for "numerical py thon". It has very efficient code for creating vectors and performing common functions (such as a dot product). So, without further ado, here's the same code in numpy. toes = np.array([8.5, 9.5, 9.9, 9.0]) wlrec = np.array([0.65, 0.8, 0.8, 0.9]) nfans = np.array([1.2, 1.3, 0.5, 1.0]) # input corresponds to every entry # for the first game of the season input = np.array([toes[0],wlrec[0],nfans[0]]) pred = neural_network(input,weight) print(pred) import numpy as np weights = np.array([0.1, 0.2, 0]) def neural_network(input, weights): pred = input.dot(weights) return pred Numpy Code Previous Code Notice that we didn't have to create a special "w_sum" function. Instead, numpy has a special function called "dot" (short for "dot product") which we can call. Many of the functions we want to use in the future will have numpy parallels, as we will see later. Both networks should simply print out: 0.98 Licensed to Asif Qamar Chapter 3 I 36 Introduction to Neural Prediction Making a Prediction with Multiple Outputs Neural Networks can also make multiple predictions using only a single input. Perhaps a simpler augmentation than multiple inputs is multiple outputs. Prediction occurs in the same way as if there were 3 disconnected single-weight neural networks. 1 An Empty Network With Multiple Outputs /* instead of predicting just whether the team won or lost, now we're also predicting whether they are happy/sad AND the percentage of the team that is hurt. We are making this prediction using only the current win/loss record */ weights = [0.3, 0.2, 0.9] def neural_network(input, weights): pred = ele_mul(input,weights) return pred input data enters here predictions come out here win loss win? sad? hurt? 2 Inserting One Input Datapoint .3 .2 .9 .3 .2 .9 65% wlrec = [0.65, 0.8, 0.8, 0.9] input = wlrec[0] pred = neural_network(input,weight) Th e most important commentary in this setting is to notice that the 3 predictions really are completely separate. Unlike neural networks with multiple inputs and a single output where the prediction is undeniably connected this network truly behaves as 3 independent compo nents, each receiving the same input data. Th is makes the network quite trivial to implement. Licensed to Asif Qamar Making a Prediction with Multiple Outputs 37 3 Perform an Elementwise Multiplication def neural_network(input, weights): pred = ele_mul(input,weights) return pred def ele_mul(number,vector): output = [0,0,0] assert(len(output) == len(vector)) for i in xrange(len(vector)): output[i] = number * vector[i] return output (0.65 * 0.3) = 0.195 = hurt prediction (0.65 * 0.2) = 0.13 = win prediction (0.65 * 0.9) = 0.585 = sad prediction inputs weights final predictions .3 .2 .9 65% .195 .13 .585 4 Deposit Predictions .3 .2 .9 65% .195 .13 .585 wlrec = [0.65, 0.8, 0.8, 0.9] input = wlrec[0] pred = neural_network(input,weight) predictions (a vector of numbers) Licensed to Asif Qamar Chapter 3 I 38 Introduction to Neural Prediction Predicting with Multiple Inputs & Outputs Neural networks can predict multiple outputs given multiple inputs. 1 An Empty Network With Multiple Inputs & Outputs #toes %win #fans weights = [[0.1, 0.1, -0.3],#hurt? [0.1, 0.2, 0.0], #win? [0.0, 1.3, 0.1]]#sad? def neural_network(input, weights): pred = vect_mat_mul(input,weights) return pred .1 .2 .0 win loss #toes #fans win? sad? hurt? 1 . 2 inputs predictions 2 Inserting One Input Datapoint .1 .2 .0 1 . 2 inputs predictions toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65,0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) /* This dataset is the current status at the beginning of each game for the first 4 games in a season. toes = current number of toes wlrec = current games won (percent) nfans = fan count (in millions) */ 8.5 65% 1.2 Finally, the way in which we built a network with multiple inputs or outputs can be combined together to build a network that has both multiple inputs AND multiple outputs. Just like before, we simply have a weight connecting each input node to each output node and pre diction occurs in the usual way. Licensed to Asif Qamar Predicting with Multiple Inputs & Outputs 39 3 For Each Output, Perform a Weighted Sum of Inputs .1 .2 .0 8.5 65% 1.2 def neural_network(input, weights): pred = vect_mat_mul(input,weights) return pred def vect_mat_mul(vect,matrix): assert(len(a) == len(b)) output = 0 for i in range(a): output += (a[i] * b[i]) return output #toes %win #fans (8.5 * 0.1) + (0.65 * 0.1) + (1.2 * -0.3) = 0.555 = hurt prediction (8.5 * 0.1) + (0.65 * 0.2) + (1.2 * 0.0) = 0.98 = win prediction (8.5 * 0.0) + (0.65 * 1.3) + (1.2 * 0.1) = 0.965 = sad prediction .85 .13 .0 hurt? win? sad? 4 Deposit Predictions .1 .2 .0 1 . 2 inputs predictions 8.5 65% 1.2 .555 .98 .965 toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65,0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) Licensed to Asif Qamar Chapter 3 I 40 Introduction to Neural Prediction Multiple Inputs & Outputs - How does it work? It performs 3 independent weighted sums of the input to make 3 predictions. I fi nd that there are 2 perspectives one can take on this architecture. You can either think of it as 3 weights coming out of each input node, or 3 weights going into each output node. For now, I fi nd the latter to be much more benefi cial. For now, think about this neural network as 3 independent dot products, 3 independent weighted sums of the input. Each output node takes its own weighted sum of the input and makes a prediction. 1 An Empty Network With Multiple Inputs & Outputs #toes %win #fans weights = [[0.1, 0.1, -0.3],#hurt? [0.1, 0.2, 0.0], #win? [0.0, 1.3, 0.1]]#sad? def neural_network(input, weights): pred = vect_mat_mul(input,weights) return pred .1 .2 .0 win loss #toes #fans win? sad? hurt? 1 . 2 inputs predictions 2 Inserting One Input Datapoint .1 .2 .0 1 . 2 inputs predictions toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65,0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) /* This dataset is the current status at the beginning of each game for the first 4 games in a season. toes = current number of toes wlrec = current games won (percent) nfans = fan count (in millions) */ 8.5 65% 1.2 Licensed to Asif Qamar Multiple Inputs & Outputs - How does it work? 41 3 For Each Output, Perform a Weighted Sum of Inputs .1 .2 .0 8.5 65% 1.2 def neural_network(input, weights): pred = vect_mat_mul(input,weights) return pred def vect_mat_mul(vect,matrix): assert(len(a) == len(b)) output = vector_of_zeros(len(vect)) for i in range(len(vect)): output[i] = w_sum(vect,matrix[i]) return output #toes %win #fans (8.5 * 0.1) + (0.65 * 0.1) + (1.2 * -0.3) = 0.555 = hurt prediction (8.5 * 0.1) + (0.65 * 0.2) + (1.2 * 0.0) = 0.98 = win prediction (8.5 * 0.0) + (0.65 * 1.3) + (1.2 * 0.1) = 0.965 = sad prediction .85 .13 .0 hurt? win? sad? As mentioned on the previous page, we are choosing to think about this network as a series of weighted sums. Th us, in the code above, we created a new function called "vect_mat_ mul". Th is function iterates through each row of our weights (each row is a vector), and makes a prediction using our w_sum function. It is literally performing 3 consecutive weighted sums and then storing their predictions in a vector called "output". Th ere's a lot more weights fl ying around in this one, but isn't that much more advanced than networks we have previously seen. I want to use this "list of vectors" and "series of weighted sums" logic to introduce you to two new concepts. See the weights variable in step (1)? It's a list of vectors. A list of vectors is simply called a matrix. It is as simple as it sounds. Furthermore, there are functions that we will fi nd ourselves commonly using that leverage matrices. One of these is called vector-matrix mul tiplication. Our "series of weighted sums" is exactly that. We take a vector, and perform a dot product with every row in a matrix**. As we will fi nd out on the next page, we even have special numpy functions to help us out. ** Note: For those of you experienced with Linear Algebra, the more formal defi nition would store/process weights as column vec tors instead of row vectors. Th is will be rectifi ed shortly. Licensed to Asif Qamar Chapter 3 I 42 Introduction to Neural Prediction Predicting on Predictions Neural networks can be stacked! 1 An Empty Network With Multiple Inputs

& Outputs -.1 .1 .9 win loss #toes #fans . 1 .1 .2 .0 win? sad? hurt? 1 . 2 inputs predictions #toes %win #fans ih_wgt = [[0.1, 0.2, -0.1],#hid[0] [-0.1,0.1, 0.9], #hid[1] [0.1, 0.4, 0.1]]#hid[2] # hid[0] hid[1] hid[2] hp_wgt = [[0.3, 1.1, -0.3],#hurt? [0.1, 0.2, 0.0], #win? [0.0, 1.3, 0.1]]#sad? weights = [ih_wgt, hp_wgt) def neural_network(input, weights): hid = vect_mat_mul(input,weights[0]) pred = vect_mat_mul(hid,weights[1]) return pred hiddens 2 Predicting the Hidden Layer hid[0] hid[1] hid[2] inputs predictions hiddens toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65,0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) def neural_network(input, weights): hid = vect_mat_mul(input,weights[0]) pred = vect_mat_mul(hid,weights[1]) return pred 8.5 65% 1.2 -.1 .1 .9 .86 .295 1.23 As the pictures below make clear, one can also take the output of one network and feed it as input to another network. Th is results in two consecutive vector-matrix multiplications. It may not yet be clear why you would predict in this way. However, some datasets (such as image classifi cation) contain patterns that are simply too complex for a single weight matrix. Later, we will discuss the nature of these patterns. For now, it is suffi cient that you know this is possible. Licensed to Asif Qamar Numpy Version 43 3 Predicting the Output Layer (and depositing the prediction) inputs predictions hiddens toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65,0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) def neural_network(input, weights): hid = vect_mat_mul(input,weights[0]) pred = vect_mat_mul(hid,weights[1]) return pred 65% 1.2 .86 .295 1.23 .1 .2 .0 .214 .145 .507 Numpy Version import numpy as np #toes %win #fans ih_wgt = np.array([[0.1, 0.2, -0.1],#hid[0] [-0.1,0.1, 0.9], #hid[1] [0.1, 0.4, 0.1]]).T #hid[2] # hid[0] hid[1] hid[2] hp_wgt = np.array([[0.3, 1.1, -0.3],#hurt? [0.1, 0.2, 0.0], #win? [0.0, 1.3, 0.1]]).T#sad? weights = [ih_wgt, hp_wgt] def neural_network(input, weights): hid = input.dot(weights[0]) pred = hid.dot(weights[1]) return pred toes = np.array([8.5, 9.5, 9.9, 9.0]) wlrec = np.array([0.65,0.8, 0.8, 0.9]) nfans = np.array([1.2, 1.3, 0.5, 1.0]) input = np.array([toes[0],wlrec[0],nfans[0]]) pred = neural_network(input,weights) print pred Licensed to Asif Qamar Chapter 3 I 44 Introduction to Neural Prediction A Quick Primer on Numpy Numpy is so easy to use that it does a few things for you. Let's reveal the magic. So far in this chapter, we've discussed two new types of mathematical tools, vectors and matrices. Furthermore, we have learned about different operations that occur on vectors and matrices including dot products, elementwise multiplication and addition, as well as vector-ma trix multiplication. For these operations, we've written our own python functions that can oper ate on simple python "list" objects. In the short term, we will keep writing/using these functions so that we make sure we fully understand what's going on inside them. However, now that we've mentioned both "numpy" and several of the big operations, I'd like to give you a quick run-down of basic "numpy" use so that you will be ready for our transition to "only numpy" a few chapters from now. So, let's just start with the basics again, vectors and matrices. import numpy as np a = np.array([0,1,2,3]) # a vector b = np.array([4,5,6,7]) # another vector c = np.array([[0,1,2,3],# a matrix [4,5,6,7]]) d = np.zeros((2,4))#(2x4 matrix of zeros) e = np.random.rand(2,5) # random 2x5 # matrix with all numbers between 0 and 1 print a print b print c print d print e [0 1 2 3] [4 5 6 7] [[0 1 2 3] [4 5 6 7]] [[0. 0. 0. 0.] [0. 0. 0. 0.]] [[0.22717119 0.39712632 0.0627734 0.08431724 0.53469141] [0.09675954 0.99012254 0.45922775 0.3273326 0.28617742]] Output We can create vectors and marices in multiple ways in numpy. Most of the common ones for neural networks are listed above. Note that the processes for creating a vector and a matrix are identical. If you create a matrix with only one row, you're creating a vector. Further more, as in mathematics in general, you create a matrix by listing (rows,columns). I say that only so that you can remember the order. Rows comes first. Columns comes second. Let's see some operations we can do on these vectors and matrices. print a * 0.1 # multiplies every number in vector "a" by 0.1 print c * 0.2 # multiplies every number in matrix "c" by 0.2 print a * b # multiplies elementwise between a and b (columns paired up) print a * b * 0.2 # elementwise multiplication then multiplied by 0.2 print a * c # since c has the same number of columns as a, this performs # elementwise multiplication on every row of the matrix "c" print a * e # since a and e don't have the same number of columns, this # throws a "Value Error: operands could not be broadcast together with.." Licensed to Asif Qamar A Quick Primer on Numpy 45 Go ahead and run all of the code on the previous page. The first big of "at first confusing but eventually heavenly" magic should be visible on that page. When you multiply two variables with the "*" function, numpy automatically detects what kinds of variables you're working with and "tries" to figure out the operation you're talking about. This can be mega-convenient but sometimes makes numpy a bit hard to read. You have to make sure you keep up with what each variable type is in your head as you go along. The general rule of thumb for anything elementwise (+,-,*,/) is that the two variables must either have the SAME number of columns, or one of the variables must only have 1 col umn. For example, "print a * 0.1" takes a vector and multiplies it by a single number (a scalar). Numpy goes "oh, I bet I'm supposed to do vector-scalar multiplication here" and then it takes the scalar (0.1) and multiplies it by every value in the vector. This looks exactly the same as "print c * 0.2", except that numpy knows that c is a matrix. Thus, it performs scalar-matrix multipli cation, multiplying every element in c by 0.2. Because the scalar has only one column, you can multiply it by anything (or divide, add, or subtract for that matter) Next up, "print a * b". Numpy first identifies that they're both vectors. Since neither vec tor has only 1 column, it checks to see if they have an identical number of columns. Since they do, it knows to simply multiply each element by each element based on their positions in the vectors. The same is true with addition, subtraction and division. "print a * c" is perhaps the most elusive. "a" is a vector with 4 columns. "c" is a (2x4) matrix. Neither have only one column, so numpy checks to see if they have the same number of columns. Since they do, numpy multiplies the vector "a" by each row of "c" (as if it was doing elementwise vector multiplication on each row). Again, the most confusing part about this is that all of these operations look the same if you don't know which variables are scalars, vectors, or matrices. When I'm "reading numpy", I'm really doing 2 things, reading the operations and keeping track of the "shape" (number of rows and columns) of each operation. It'll take some practice, but eventually it becomes second nature. a = np.zeros((1,4)) # vector of length 4 b = np.zeros((4,3)) # matrix with 4 rows & 3 columns c = a.dot(b) print c.shape (1,3) Output There is one golden rule when using the 'dot' function. If you put the (rows,cols) de scription of the two variables you're "dotting" next to each other, neighboring numbers should always be the same. In this case, we're dot producting a (1,4) with a (4,3). Thus, it works fine, and outputs a (1,3). In terms of variable shape, you can think of it this way. Regardless of wheth- (a,b).dot(b,c) = (a,c) er you're "dotting" vectors or matrices. Their "shape" (number of rows and columns) must line up. The col umns on the "left" matrix must equal rows on the "right". Licensed to Asif Qamar Chapter 3 I 46 Introduction to Neural Prediction Conclusion To predict, neural networks perform repeated weighted sums of the input. We have seen an increasingly complex variety of neural networks in this chapter. I hope that it is clear that a relatively small number of simple rules are simply used repeatedly to create larger, more advanced neural networks. Furthermore, the intelligence of the network really de pends on what weight values we give to our networks. In the next chapter, we will be learning how to set our weights so that our neural networks make accurate predictions. We will find that in the same way that prediction is actu ally based on several simple techniques that are simply repeated/stacked on top of each other, "weight learning" is also a series of simple techniques that are simply combined many times across an architecture. See you there!